Transformers

Introduction

In this project we explore the Visual Transformer (ViT), by implementing a crafting a simplified, smaller version of it.

We first implemented the fundamentals of the ViT from scratch, creating components such as patch embeddings, self-attention mechanisms, multi-head self-attention, and constructing transformer blocks.

Secondly, we re-implemented the ViT using the Timm library and evaluated the network on MNIST datasets. We also compared the network with pretrained weights (on ImageNet) and without.

Linear Projection of flattened patches

Q1: First, we need to code the patch embedding. In NLP, it could be using a word2vec embeddings, but we need something different in Computer Vision.

In the code

Q2: MLP Now we need to build a transformer block. Let's first build the easiest part, the MLP! By now, you should be confident how to code one. Code a MLP with one hidden layer. Note that the input and output dimensions will be the same. Instead of ReLU, use the activation GELU, which is a slight alternative often used for transformers.

In the code

Q3: Self-attention. Now we are going to build the famous Self-Attention. What is the main feature of self-attention, especially compared to its convolutional counterpart. What is its main challenge in terms of computation/memory? At first, we are going to only consider the simple case of one head. Write the equations and complete the following code. And don't forget a final linear projection at the end!

The main feature of self-attention is to global dependencies within a sequence: it allows each element in a sequence to focus on other elements and determine their importance or relevance. One advantage compared to the convolutional counterpart is that self-attention does not have fixed patterns or kernel sizes. This leads transformers to be more flexible than the standard CNN. The main challenge of the transformers is the computational and memory cost, since it is directly proportional to the quadratic of the sequence length.

Attention equation:
Given,

$$Q = X\,W_Q \in \mathbb{R}^{T \times d_K}\ (\text{Query})$$

$$K = X\,W_K \in \mathbb{R}^{T \times d_K}\ (\text{Key})$$

$$V = X\,W_V \in \mathbb{R}^{T \times d_V}\ (\text{Value})$$

$$T : \text{sequence length}$$

$$d_K, d_V : \text{hidden (embedding) dimension of queries/keys/ values, here: } d_K = d_V$$

Where X is the input sequence of embeddings; Wq, Wk and Wv are the weight matrixes to transform the input sequence into query, key and value vector.

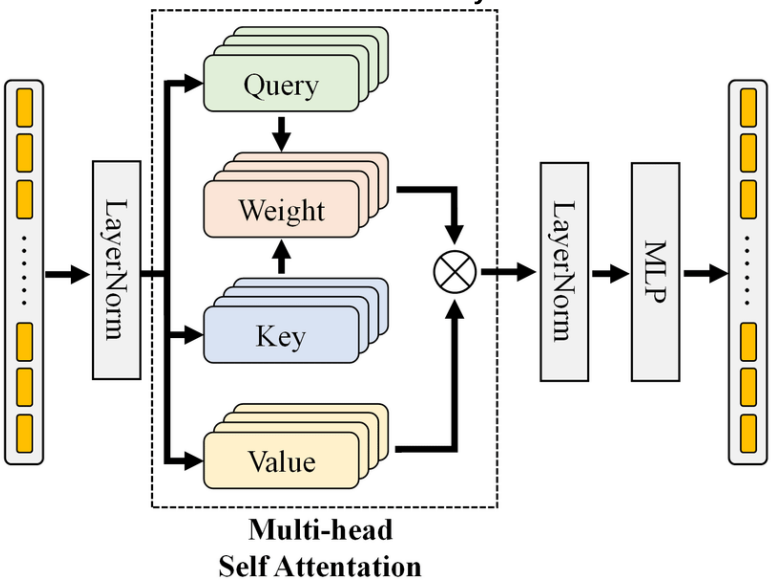
$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

Q4 : Multi-head self-attention. Now we need to build a Multi-Heads Self-Attention. Write the equations and complete the following code to build a Multi-Heads Self-Attention.

$$\text{MultiHeadAttention}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^0$$

$$\text{head}_i = \text{self-attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Vizualization of Multi-Head Attention Layer



- The total embedding size will be shared equally among all heads.

Q5: Transfomer block. Now, we need to build a Transformer Block as described in the image below. Write the equations and complete the following code.

Normalization layer 1:

$$LayerNorm1(X + Multihead(X))$$

Where X is the input sequence of embeddings;
Feed-Forward:

$$FFN(X) = GeLU(X \cdot W1 + b1) \cdot W2 + b2$$

Normalization layer 2:

$$LayerNorm2(X + FFN(X))$$

Q6: Full ViT model

Now you need to build a ViT model based on what you coded in the previous questions. There are additional components that should be coded such as the Class token, Positional embedding and the classification head.

a) Explain what is a class token and why we use it?
➤ The "class token" is a unique token added to the start of the input sequence. Initially set with random values, it undergoes training through backpropagation, adjusting its weight vector. Throughout the transformer network, it is repeatedly re-embedded.
➤ Arriving at the conclusion of the sequence, it serves the purpose of classification. The class token gathers and retains significant information. This concept is inspired by Bert, a prominent NLP model.
➤ In this setup, the MLP works only on the learned class tokens.
b) Explain what is the the positional embedding and why it is important?
➤ Multi-head attention layer is permutation equivariant (does not change with changed word/patch order). Therefore, we have to pass additional clues to the model to respect the word order within a sentence.
We use it so as not to loose the information about the spatial position of the patches
c) Test different hyperparameters and explain how they affect the performance. In particular embed_dim, patch_size, and nb_blocks.
➤ For PE, you can use a sinusoidal encoding (see below), or fully learned.

Rewrite positional encoding formula

$$\mathrm{PE}(k,i) = \begin{cases} \sin\left(\frac{\text{position}}{10000^{\frac{i}{d}}}\right) \text{ if } i \text{ even} \\ \cos\left(\frac{\text{position}}{10000^{\frac{(i-1)}{d}}}\right) \text{ else} \end{cases} \Leftrightarrow \mathrm{PE}(k,2i) = \sin\left(\frac{\text{position}}{10000^{\frac{2i}{d}}}\right) \mathrm{PE}(k,2i+1) = \cos\left(\frac{\text{position}}{10000^{\frac{2i}{d}}}\right)$$

$$PE_{(k,2i)} = sin(\frac{pos}{10000^{\frac{2i}{d_{model}}}})$$

$$PE_{(k,2i+1)} = cos(\frac{pos}{10000^{\frac{2i}{d_{model}}}})$$

$d\_model = 4$

$$k \begin{cases} P(0) = \left[sin(\frac{0}{10000^{\frac{0}{4}}}), cos(\frac{0}{10000^{\frac{0}{4}}}), sin(\frac{0}{10000^{\frac{1}{4}}}), cos(\frac{0}{10000^{\frac{1}{4}}})\right] \\ P(1) = \left[sin(\frac{1}{10000^{\frac{0}{4}}}), cos(\frac{1}{10000^{\frac{0}{4}}}), sin(\frac{1}{10000^{\frac{1}{4}}}), cos(\frac{1}{10000^{\frac{1}{4}}})\right] \\ P(2) = \left[sin(\frac{2}{10000^{\frac{0}{4}}}), cos(\frac{2}{10000^{\frac{0}{4}}}), sin(\frac{2}{10000^{\frac{1}{4}}}), cos(\frac{2}{10000^{\frac{1}{4}}})\right] \\ P(3) = \left[sin(\frac{3}{10000^{\frac{0}{4}}}), cos(\frac{3}{10000^{\frac{0}{4}}}), sin(\frac{3}{10000^{\frac{1}{4}}}), cos(\frac{3}{10000^{\frac{1}{4}}})\right] \\ P(4) = \left[sin(\frac{4}{10000^{\frac{0}{4}}}), cos(\frac{4}{10000^{\frac{0}{4}}}), sin(\frac{4}{10000^{\frac{1}{4}}}), cos(\frac{4}{10000^{\frac{1}{4}}})\right] \\ P(5) = \left[sin(\frac{5}{10000^{\frac{0}{4}}}), cos(\frac{5}{10000^{\frac{0}{4}}}), sin(\frac{5}{10000^{\frac{1}{4}}}), cos(\frac{5}{10000^{\frac{1}{4}}})\right] \end{cases}$$

i=0 i=0 i=1 i=1

Experimental analysis (MNIST)

Q7: What is the complexity of the transformer in terms of number of tokens? and how you can improve it (you check latest papers on this topic)

The complexity of the transformers in terms of number of tokens is $O(L * n^2)$ where L is the number of layers used and n is the sequence length (number of tokens). Therefore the complexity of the transformer is quadratic relative to the number of tokens.

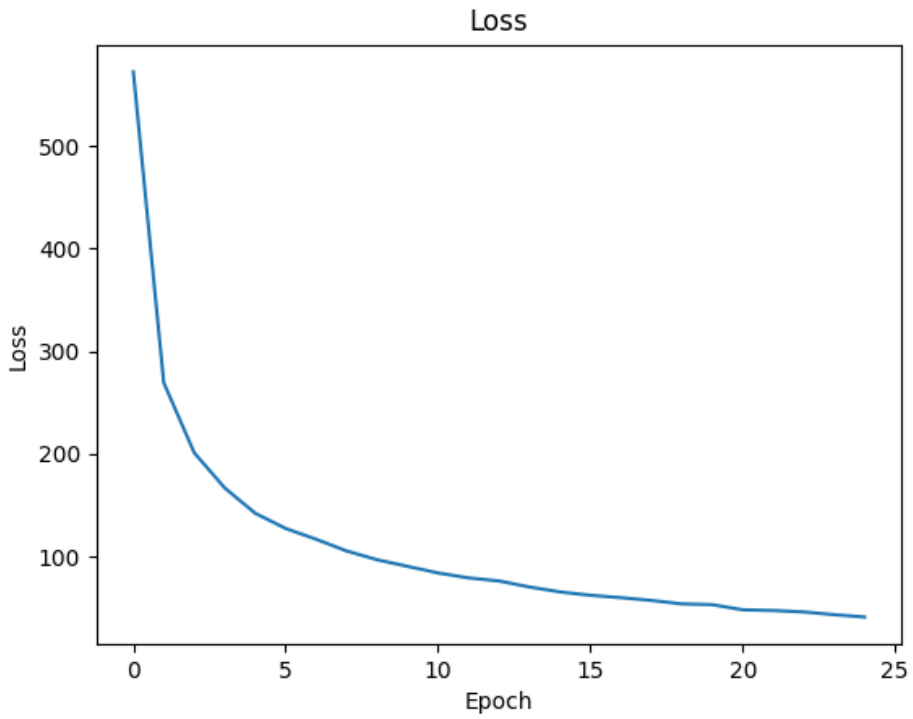
To improve the computational cost we could use:

1. appropriate initialization and optimization paradigms that can accelerate the convergence rate with fewer training iterations, resulting in lower computational costs;
2. higher data efficiency by sampling informative training samples towards more efficient neural scaling laws of test error with respect to dataset size;
3. memory-efficient techniques to meet the memory requirements for training large Transformers, which requires jointly optimizing PE utilization, memory and communication footprints across accelerators, using parallelism, low-precision arithmetic, checkpointing and of-floading strategies, etc;
4. hardware and algorithm co-design to maximize the training scalability on hardware platforms.
   source: "A Survey on Efficient Training of Transformers", https://arxiv.org/pdf/2302.01107.pdf
   Another interesting approach is to use techniques of merging and and pruning the tokens (source: "Learned Thresholds Token Merging and Pruning for Vision Transformers", https://arxiv.org/pdf/2307.10780.pdf)

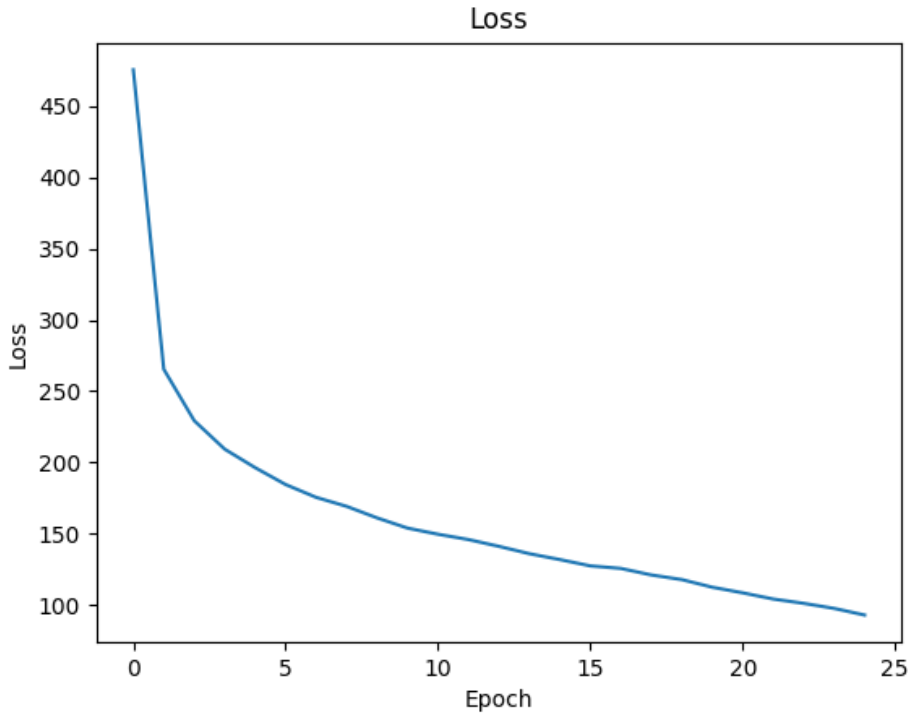
Q8: Try to use bigger transformer, for example the ViT-S from the timm library. Test with and without intialization from imagenet.

a) Load the model using the timm library without pretrained wewights. Try to apply it directly on a tensor with the same MNIST images resolution. What it is the problem and why we have it? Explain if we have also such problem with CNNs. As ViT takes RGB images, the input tensor should have 3 channels.
→ Input shape is fixed for CNNs and ViTs by choice of architecture. We can go around by rezising (shrinking or interpolating) but we are not ideally using either the image details or the efficiency of the CNN. (same for channels)

b) There is a trick in timm that allows to use pretrained models with different image resolution and number of classes. Try to reload the model to be able to use on MNIST images:
c) redo the training with the pretrained ViT-S



c) redo the training but with the ViT-S pretrained on ImageNet



d) Comment the final results and provide some ideas on how to make transformer work on small datasets. You can take inspiration from some recent work.
--> Transformers as many NNs are overparametrized for small datasets and we have to rely on regularization techniques (e.g. early stopping, dropout,...) or increase the number of training samples (e.g. training augmentations, pre-training on a different dataset and finetuning afterwards (transfer learning),...)